the Compiler **42** will automatically generate an indirection. The linker will catch the indirect reference and provide a local address which will be patched with the external address at load time.

Data-to-Code & Data-to-Data

Example (Data-to-Code): void (*pfn)( ) = Foo;

Example (Data-to-Data): int& pi = i;

Since both of these references require absolute addresses, they will be handled during loading. The patching of data references at load time will be handled just like the patching of external references.

FIG. **42** shows what happens in each type of reference. All of these cases show the internal usage case. If an external library references these same components, this library will receive several GetExportAddress( ) calls at load time. In response to the GetExportAddress( ), a library will return the internal linkage area address for functions, and the real address for data. This allows the functions to move around while the library is loaded.

Linkage Areas

The internal linkage area is completely homogeneous (each entry is: JMP address). The external area has different types of entries. A normal function call will have a jump instruction in the linkage area, while a virtual function call will have a thunk that indexes into the virtual table. Pointers to member functions have a different style of thunk.

While the invention has been described in terms of a preferred embodiment in a specific programming environment, those skilled in the art will recognize that the invention can be practiced with modification within the spirit and scope of the appended claims.

Having thus described our invention, what we claim as new and desire to secure by Letters Patent is as follows:

1. In a computer system having a memory, a display, a program counter with a value, a program consisting of a set of named components stored in a database in the memory, each component including an attribute indicating whether the component data is valid, source code for implementing the component, and object code for executing the component, and a debugger for monitoring execution of the program during debugging to detect a program execution halt, a method for dynamically generating symbolic debugging information, comprising the steps of:

(a) when program execution halts during debugging, using the program counter value to locate a component in the database;

(b) checking the attribute of the located component to determine whether symbolic debugging information relating the object code to the source code is valid;

(c) generating the symbolic debugging information by recompiling the source code of the located component when the symbolic debugging information is not valid;

(d) associating valid symbolic debugging information with the located component; and

(e) using the debugger to continuing debugging the program.

2. The method of claim **1**, wherein step (a) comprises the steps of:

(a1) using the program counter value to address a cache memory; and

(a2) obtaining a component name from the cache memory.

3. The method of claim **2**, wherein step (a) comprises the steps of:

(a3) converting the program counter value into a component name when the cache memory does not contain a component name at a location addressed by the program counter value; and

(a4) storing the component name determined in step (a3) in the cache memory.

4. The method of claim **1**, wherein step (c) comprises the step of:

(c1) using the compiler to create, as part of the symbolic debugging information, at least one map which indicates a relation between the component object code and the component source code; and

(c2) associating the at least one map with the located component.

5. The method of claim **1**, wherein each computer program is constructed as a collection of components with dependencies between components, each component having an interface and an implementation and wherein all component dependencies are from component interfaces and wherein step (c) comprises the step of:

(c3) recompiling source code for the located component and all components which depend on the located component.

6. The method of claim **1**, wherein step (c) comprises the step of:

(c4) updating symbolic debugging information which was originally created by compiling all of the components.

7. The method of claim **1**, further comprising the step of:

(f) executing a browser program with the symbolic debugging information generated in step (c) to present source code from a located component on the display when program execution halts during debugging in response to an exception generated by the program.

8. The method of claim **7**, wherein step (f) comprises the step of:

(f1) applying a program thread which generated the exception as an input to the browser program.

9. Apparatus for dynamically generating symbolic debugging information for use in a computer system having a memory, a display, a program counter with a value, a program consisting of a set of named components stored in a database in the memory, each component including an attribute indicating whether the component data is valid, source code for implementing the component, and object code for executing the component, and a debugger for monitoring execution of the program to detect a program execution halt during debugging, the apparatus comprising:

(a) means responsive to a program execution halt during debugging, for using the program counter value to locate a component in the database;

(b) means responsive to the attribute of the located component for determining whether symbolic debugging information relating the object code to the source code is valid;

(c) means for controlling a compiler to generate the symbolic debugging information by recompiling the source code of the located component when the symbolic debugging information is not valid;

(d) means for associating valid symbolic debugging information with the located component; and

(e) means for controlling the debugger to continuing debugging the program.

10. The apparatus of claim **9**, wherein the means for using the program counter value to locate the component comprises a cache memory for storing at least one component name and means for using the program counter value to address the cache memory to obtain a component name from the cache memory.